# VORTEX++ Class Reference Manual

May 30, 2017

Trifox Inc.

www.trifox.com

## Trademarks

TRIMapp, TRImpl, TRIMqmr, TRIMreport, TRIMtools, GENESISsql, DesignVision, DVapp, DVreport, VORTEX, VORTEXcli, VORTEXc, VORTEXcobol, VORTEXperl, VORTEXjdbc, VORTEX++, VORTEXJava Edition, LIST Manager, VORTEXodbc, VORTEXnet, VORTEXclient/server, VORTEXaccelerator, VORTEXreplicator are all trademarks of Trifox, Inc.

All other brand and product names are trademarks or registered trademarks of their respective owners.

## Copyright



Printed in the U.S.A.

# Contents

# Preface

## Background

Trifox Inc. has been serving the relational database market since 1984 through consulting and the development of software products. In 1987, Trifox created SQL*QMX for Oracle. This easy-to-use, powerful querying and report writing tool, which is based on IBM's QMF, continues to be used at thousands of sites. In 1989, Trifox created TRIMtools, a family of application and reportwriting tools now known as DesignVision. DesignVision was developed in response to the OLTP requirements of several large application vendors.

## Database Access

VORTEX is an integrated family of products that allows nearly any production application to access SQL data:

- On any or all of the major relational databases.
- Across networks.
- Across platforms.
- With a dramatic increase in the number of concurrent users.
- Without any additional hardware.

In a client/server or multi-tier configuration, VORTEX makes it possible for your SQL applications to access data on different platforms over one or more network configurations. Currently it supports only TCP/IP.

Inherent in this approach are services that allow production applications originally written for one relational database (such as Oracle) can access the same data on another database (such as Informix), even if it is spread across different databases.

VORTEX Precompilers for C and COBOL, as well as a variety of program interfaces, allow existing SQL programs to take full advantage of VORTEX services such as performance enhancement, transaction monitoring, and flat-file database access.

With VORTEXaccelerator in your configuration, you dramatically increase the number of concurrent users who can log on to a specific SQL production application. Your users experience faster performance and you won't have to change any programs or add any hardware.

## Application and Report Development

DesignVision DVapp lets you design, generate, and maintain forms-based applications. You can easily port the pop-up windows, customizable menus and submenus, and

custom keyboard assignments, in fact the entire application, to Windows .NET, Unix, OpenVMS, or HTML5 with no extra effort.

The reportwriter, TRIMreport, lets you create simple reports quickly, or complex reports with absolute confidence in their power.

When you want to write stand-alone applications (including triggers) without a user interface, the TRIMpl 4GL language gives you the freedom you want. The procedural language has over 100 database-specific functions that help you write powerful applications in very little time.

## Reaching Legacy Data

GENESISsql is a SQL processor that accesses low-level data sources such as ISAM, SDMS, ADABAS, RMS, and MicroFocus and makes the data accessible to VORTEX clients. You can add GENESIS data sources to a VORTEX system in a matter of days, simplifying what used to be an enormous task.

## Conventions

Screen shots in this manual come from the Windows version of our software.

Trifox documentation uses the following conventions for communicating information:

| Example | Describes |
|---|---|
| `CHOOSE REPORT > [F3] >` | Press [F3] on the CHOOSE REPORT menu and ... |
| Right-click | Clicking the right mouse button. |
| Left-click | Clicking the left mouse button. |
| *`connect_string`* | Replace italicized text with your own variable. |
| **`vtxnetd`** | Text in bold typewriter style represents strings that you type exactly as they appear in the manual. |

## Support

If you have a question about a TRIFOX product that is not answered in the documentation (paper or online), contact the Customer Support Services group at:

- support@trifox.com
- Trifox Customer Support Services
  2959 Winchester Boulevard
  Campbell, CA 95008
  U.S.A.
- 408-796-1590

# Revisions

*August 1996*

First release.

# Chapter 1 Overview

## Overview

VORTEX++ is a thin class library that provides a method of accessing the VORTEX API, which is a message-based interface. The interface is message-based, machine-independent, and supports three data types:

- CHAR/BINARY
- NUMBER
- DATETIME

Since the API is message based and machine-independent, it moves data very efficiently but isn't easy to use. VORTEX++ enhances usability by providing:

- A few simple classes that handle all database access.
- Convenient classes to handle the basic data types.
- Operator overloading.
- Conversion methods.

The main VORTEX++ classes are:

- ***dbChannel*** — Encapsulates a database connection. Each instance corresponds to an actual database connection.
- ***dbStatement*** — Encapsulates a SQL statement. Each instance corresponds to a logical VORTEX cursor.
- ***dbDescriptor*** — Describes an output column.
- ***dbException*** — The base class for all VORTEX++ exceptions.
- ***dbObject*** — The base class for all VORTEX++ data objects.

# Chapter 2 dbChannel

## class dbChannel

Encapsulates a database connection.

## Data Members

```
int typeOfLastException;
```

`typeOfLastException` is initialized to zero ( 0 ) on instantiation and is never reset. Each time VORTEX throws an exception, its type is stored in this information variable. The possible values are:

1. dbException
2. dbExceptError
3. dbExceptDB
4. dbExceptUser
5. dbExceptFatal
6. dbExceptVortex
7. dbExceptNet

See `dbChannel.hxx` for more details.

## Constructors

```
dbChannel ( const int maxLogicalCursors,
            const int maxDbCursors,
            const int maxColumns,
            const int fetchBufferSize);
```

**maxLogicalCursors** Maximum number of logical cursors to be used. Set this to some high number, such as 512.

**maxDbCursors** Maximum number of actual database cursors to allocate. A good number is 1/8 of `maxLogicalCursors.`

**maxColumns** Maximum number of columns to be processed at any given time.

**fetchBufferSize** Size of the internal buffer used to pre-fetch data.

## Destructor

A database release is automatically performed if a connection is active.

```
~dbChannel();
```

# Exceptions

The following exceptions can be thrown by dbChannel functions.

dbExceptionError

Missing parameter, row: n, column: m

Datatype mismatch, row: n, column: m

Unknown BIND type n

Invalid execute procedure state

Data conversion failed

Invalid method for procedure

Invalid execute cursor state

Invalid fetch cursor state

Invalid SELECT statement

Expected a dbBinary

Expected a dbChar

Expected a dbNumber

Expected a dbDate

Unexpected datatype n

returnObject cannot be NULL

dbExceptDB

Any message that can be returned from the target DBMS

dbExceptUser

Any message that can be returned from the target DBMS and is remapped to a user defined error number

dbExceptFatal

No more logical cursors available

dbExceptVortex

See the Codes and Messages chapter of the VORTEX Installation and Usage Guide, VORTEX Error Messages section.

dbExceptNet

Any operating system network error message

# Member Functions

These functions are listed in `dbChannel.hxx`.

## cancel

Cancels any outstanding database request. Typically this function is called from a Ctrl-C handler when the user wants to interrupt a wild query.

```
void cancel();
```

## command

Executes a database driver-specific command.

```
void command (dbStatement *theStatement,
              const int   commandCode,
              const char *commandParms);
```

**theStatement** NULL or, if a statement (cursor) command, a previously instantiated `dbStatement`. The parameter is ignored for commands that are session-oriented.

**commandCode** The command code.

**commandParms** A null-terminated string containing the command parameters. Multiple parameters must be blank-separated.

The following command are available: (see `dbChannel.hxx` for definitions.)

1. TDB_CMD_TIMEOUT
2. TDB_CMD_SINGLEPID
3. TDB_CMD_OOPT
4. TDB_CMD_USEDB
5. TDB_CMD_LANGUAGE
6. TDB_CMD_RAW_DATETIME
7. TDB_CMD_VERSION
8. TDB_CMD_CHAR

9. TDB_CMD_LOCK
10. TDB_CMD_ONEBYTE
11. TDB_CMD_MULTISTMT
12. TDB_CMD_AUTOCOMMIT
13. TDB_CMD_DORMANT
14. TDB_CMD_GETLDA
15. TDB_CMD_RAISE
16. TDB_CMD_ERROR
17. TDB_CMD_TMP_RELEASE
18. TDB_CMD_WORM_AUTH
19. TDB_CMD_TIMEOUT_QUERY
20. TDB_CMD_TIMEOUT_FETCH
21. TDB_CMD_DYN_CURSOR
22. TDB_CMD_SYNTAX_LEVEL
23. TDB_CMD_RETURN_ROWID
24. TDB_CMD_NEW_BLOBS
25. TDB_CMD_RO_CURSOR
26. TDB_CMD_CURSOR_TYPE
27. TDB_CMD_TXN_ISOLEVEL
28. TDB_CMD_SCROLL
29. TDB_CMD_HOSTNAME
30. TDB_CMD_FQCOLNAME
31. TDB_CMD_FREE_LOCKS
32. TDB_CMD_NOUNICHAR
33. TDB_CMD_ZEROLEN_NULL
34. TDB_CMD_EXIO_RS
35. TDB_CMD_DRV_CONFIG

## commit/rollback

Commits or rolls back a transaction and, optionally, starts a new read/write transaction.

```
void commit   (const int startUpdateTrans);
void rollback (const int startUpdateTrans);
```

**startUpdateTrans** If true then start a read/write transaction following the commit/rollback.

## sql

Associates a SQL statement with a `dbStatement`. If the statement contains parameters in the Oracle format `:n`, then it's translated for the target database (example: "?" for DB2). No other translation is performed.

Use `dbStatement:copyParam` and `dbStatement::setParam` to populate the parameters.

```
void sql(dbStatement  *theStatement,
         const char   *sqlStatement);
void sql(dbStatement  *theStatement,
         const char   *sqlStatement,
         const int     numParameters);
void sql (dbStatement *theStatement,
          const char  *sqlStatement,
          const int    numDimensions),
          const int    numParameters)
```

**theStatement** A previously instantiated `dbStatement`.

**sqlStatement** A null-terminated string containing the SQL statement.

**numParameters** Number of parameters in the SQL statement. If not specified, then the statement is assumed to have no parameters.

**numDimensions** Number of dimensions of parameters in the SQL statement. If not specified and `numParameters` is specified, the value defaults to 1.

## proc

Associates a stored procedure statement with a dbStatement. Note that not all databases support stored procedures.

Use `dbStatement::setHostVar` to populate the parameters.

```
void proc(dbStatement *theStatement,
          const  char *procStatement,
          const  int   numParameters);
```

**theStatement** A previously instantiated `dbStatement`.

**procStatement** A null-terminated string containing the stored procedure statement.

**numParameters** Number of parameters in the stored procedure statement. If not specified, then the statement is assumed to have no parameters.

## execute

Executes a non-SELECT statement.

```
void execute (dbStatement *theStatement);
void execute (dbStatement *theStatement,
              const int   ignoreDbError);
```

## fetch

Executes the first fetch for a SELECT statement, which fills up the fetch buffer and positions the cursor at the first record. Calls to `dbChannel::nextObject` and `dbChannel::skipObject` automatically fetch more data from the database as needed.

```
void fetch(dbStatement *theStatement);
```

**theStatement** A previously instantiated `dbStatement`.

## close

Closes a statement (cursor). For best performance, the statement should only be "softly" closed.

```
void close(dbStatmeent *theStatement,
           const int   hardClose);
```

**theStatement** A previously instantiated `dbStatement`.

**hardClose** Do an actual database cursor close.

## nextObject

Retrieves the next column from a fetch statement.

```
dbObject *nextObject(dbStatement *theStatement);
int       nextObject(dbStatement *the Statement,
                     dbObject       *returnObject,
                     int             maxLength);
```

**theStatement** A previously instantiated `dbStatement`.

**returnObject** Where to return the data. If not a CHAR data type, it must match the data type of the object to fetch.

**maxLength** Maximum length allowed by `returnObject`.

Returns

1. The `dbObject`. This is allocated from the heap and the caller is responsible for freeing it.
2. Status (length):
   -1 — NULL data.

0 — All is fine
> 0 — Data truncated to this length.

## skipObject

Skips objects from a fetch statement.

```
void skipObject(dbStatement *theStatement,
                int         numColsToSkip,
                const int   stopOnColZero);
```

**theStatement** A previously instantiated `dbStatement`.

**numColsToSkip** The number of columns to skip. If `stopOnColZero` is false, then this wraps around.

**stopOnColZero** Forces a stop at the first column of the next fetch if `numColsToSkip` results in a wrap.

## getBlob

Gets a BLOB (binary large object) from a fetch statement. You must call nextObject before calling getBlob. The nextObject call will return the total length of the blob or clob column.

```
unsigned int getBlob(dbStatement *theStatement,
                int              column,
                char            *buffer,
                unsigned int     bufferLength);
```

**theStatement** A previously instantiated `dbStatement`.

**column** The zero-based column index of the column containing the BLOB.

**buffer** Destination buffer.

**bufferLength** Destination buffer length.

Returns

The length of the BLOB portion fetched. A zero indicates that there is no more BLOB data to fetch from this column.

## putBlob

Puts (sends) a BLOB to the database. The data is not stored in the database until the `dbChannel::execute` has been performed.

```
void putBlob(dbStatement *theStatement,
             int          isBinary,
             char        *buffer,
             unsigned int  bufferLength);
```

## getRowsAffected

Returns the number of database rows affected by the last INSERT, UPDATE or DELETE operation. For all other operations, this function returns a ZERO.

```
int getRowsAffected() const;
```

## getCode/getMessage

Returns the last error code and error message.

```
int getCode() const;
char *getMessage() const;
```

## set/getIn/OutDateMask

Sets or retrieves the default DATETIME format masks to use. See Appendix A for details.

```
void  setInDateMask(const char *mask);
char *getInDateMask();
void  setOutDateMask(const char *mask);
char *getOutDateMask();
```

**mask** A null-terminated string containing the format mask.

## setUserErrorMap

Sets an array of database error codes that when encountered, throw a `dbExceptUser` rather than a `dbExceptDB`.

```
void setUserErrorMap (int       *errors,
                      const int  numErrors);
```

**errors** An array of integers containing error codes.

**numErrors** Number of error codes in `errors`.

## userError

Returns true if an error code is in the error code map. If not, returns false.

```
int userError(const int errorCode);
```

## stmtCacheOn/Off

Turns the `dbStatement` cache on or off. The default is off. Once turned on, a `dbStatement` that is destroyed is placed in the internal cache. When instantiating a new `dbStatement` (with a SQL statement), the cache is searched for a match on the SQL statement. If there is a match, the cache is used. If the `dbStatement` is instantiated without a SQL statement and is later assigned a SQL statement, the cache is not searched.

When turning off the cache any `dbStatement`s in the cache are destroyed.

```
void stmtCacheOn();
void stmtCacheOff();
```

## throwException

This is the only function that throws any `dbExceptions`.

It is a virtual function that allows users to supply their own functions if so needed. The user-supplied function must throw its own exception or call `throwException` to avoid unexpected behavior.

```
virtual void throwException(const int exceptionType,
                            const char *errorMessage);
```

**exceptionType** Type of exception to be thrown.

**errorMessage** Error message to set in exception.

## connect

Connects to an actual database.

```
void connect (char *connectString);
```

`connectString` VORTEX connect string.

## release

Releases (disconnects) from a database.

```
void release();
```

# Chapter 3 dbChar/dbBinary

## class dbChar:public dbObject

Encapsulates character or binary data. This is a sub-class derived from `dbObject`.

## Data Members

None.

## Constructors

The first constructor is used to create a NULL object. The last constructor is used to alloate a buffer without initializing it to any specific data (useful for `dbChannel::nextObject`).

```
dbChar();
dbChar(dbChar   *value);
dbChar(char     *value);
dbChar(void     *value,
       const int length);
dbChar(const int length);

dbBinary();
dbBinary(dbBinary      *value);
dbBinary(unsigned char *value,
         const    int   length);
dbBinary(const    int   length);
```

## Destructor

Frees the internally allocated data buffer.

```
~dbChar();

~dbBinary();
```

## Member Functions

See `dbObject`.

# Chapter 4
# dbDate

## class dbDate : public dbObject

Encapsulates an internal datetime. The internal format is compatible with Oracle's internal datetime. This is a sub-class derived from `dbObject`.

## Data Members

None.

## Constructors

The first constructor is used to create a NULL datetime.

## Destructor

Default.

## Exceptions

The following exceptions can be thrown by dbDate functions:

dbException

dbDate(): Invalid inDateMask

dbDate(): Invalid parameters

## Member Functions

See `dbObject` for additional functions.

### timestamp/today

Set the value to the current date. Additionally, `timestamp` sets the current time.

### toChar

Convert a datetime to a null-terminated string.

# Chapter 5
# dbDescriptor

## class dbDescriptor

A simple class that describes the output column.

## Data Members

```
int  type;
int  length;
int  nullsAllowed;
int  precision;
int  scale;
char name[31];
```

| | |
|---|---|
| **type** | Output column's data type. See `dbDescriptor.hxx` for values. |
| **length** | Output column length. |
| **nullsAllowed** | True if nulls are allowed. |
| **precision** | Precision of a numeric output column. |
| **scale** | Scale of a numeric output column. |
| **name** | Name of the output column, null-terminated. |

## Constructors

Internal.

## Destructor

`Default.`

## Member Functions

### isLargeObject

Return true of the column is a BLOB.

```
int isLargeObject() const;
```

# Chapter 6 dbException

## class dbException:public xmsg

The base class for all VORTEX++ exceptions. The derived classes are:

- `dbExceptionFatal` — Fatal VORTEX++ errors.
- `dbExceptionVortex` — Fatal VORTEX errors.
- `dbExceptionError` — Non-fatal errors in the VORTEX++ libraries.
- `dbExceptionDB` — Database errors.
- `dbExceptionNet` — Network errors.
- `dbExceptionUser` — User-defined database errors. See `dbChannel::setUserErrorMap`.

### Catching Messages

When you catch a dbException, be sure to get a **reference** to the dbException intead of a **copy** of it. Since xmsg contains a message pointer, each copy of dbException points to the same message which leads to problems when the dbException is destroyed.

***Example*** (note the "&")

```
try {
…
}
catch (dbException& e) {
     cout << "\n***** ERROR *****\n" << e.why() << "\n";
     }
```

## Data Members

```
int errorType;
```

Each time an exception is thrown, the type of the exception is stored in this information variable. The values are (see `dbChannel.hxx`):

1. typeException
2. typeExceptError
3. typeExceptDB
4. typeExceptUser
5. typeExceptFatal
6. typeExceptVortex

7. typeExceptNet

```
int errorCode;
```

Each time an exception is thrown, the error code of the exception is stored in this information variable.

# Constructors

```
dbException(dbChannel  *theChannel,
            const char *errorMessage);
```

**theChannel** A previously instantiated `dbChannel`.

**errorMessage** A null-terminated string containing the error message.

# Member Functions

## getType/getCode

Return the error type and code associated with the exception.

```
int getType() const;
int getCode() const;
```

## getChannel

Return the `dbChannel` that threw the exception.

```
dbChannel *getChannel() const;
```

# Chapter 7
# dbInt

## class dbInt:public dbObject

Encapsulates an integer. This is a sub-class derived from `dbObject`.

## Data Members

```
int data;
```

**data** The actual integer. It is safe to manipulate this directly.

## Constructors

The first constructor is used to create a NULL integer.

```
dbInt();
dbInt(dbInt     *value);
dbInt(const int  value);
dbInt(const int  value,
      const int  datatype);
dbInt(void      *value,
      const int  length);
```

**value** The value.

**datatype** For setting BLOB integer type.

**length** For integers of length less than `sizeof(int)`.

## Destructor.

Default.

## Member Functions

See `dbObject`.

# Chapter 8
# dbNumber

## class dbNumber:public dbObject

Encapsulates an internal number. The internal format is compatible with Oracle's internal numbers. This is a sub-class derived from `dbObject`.

## Data Members

None.

## Constructors

The first constructor is used to create a NULL number.

```
dbNumber();
dbNumber(dbNumber          *value);
dbNumber(const          int value);
dbNumber(const unsigned int value);
dbNumber(const          long long value);
dbNumber(const unsigned long long value);
dbNumber(void              *value,
         const          int length);
```

## Destructor.

Default.

## Member Functions

See `dbObject` for additional functions.

### toInt

Convert a number to an integer. If no parameter is given and the number is NULL, then a 0 is returned.

```
int toInt();
int toInt(int   ifNull);
int toInt(int  *status);
```

**ifNull** Value to return if the number is NULL.

**status** Status of conversion:
- -2 NULL
- -1 truncated or negative
- 0 ok
- 1 overflow

## toInt8

Convert a number to a long long integer. If no parameter is given and the number is NULL, then a 0 is returned.

```
long long toInt8();
long long toInt8(long long ifNull);
long long toInt8(int      *status);
```

**ifNull** Value to return if the number is NULL.

**status** Status of conversion:
- -2 NULL
- -1 truncated or negative
- 0 ok
- 1 overflow

## toUInt

Convert a number to an unsigned integer. If no parameter is given and the number is NULL, then a 0 is returned.

```
unsigned int toUInt();
unsigned int toUInt(int ifNull);
unsigned int toUInt(int *status);
```

**ifNull** Value to return if the number is NULL.

**status** Status of conversion:
- -2 NULL
- -1 truncated or negative
- 0 ok
- 1 overflow

## toUInt8

Convert a number to an unsigned integer. If no parameter is given and the number is NULL, then a 0 is returned.

```
unsigned long long toUInt8();
unsigned long long toUInt8(long long ifNull);
unsigned long long toUInt8(int      *status);
```

**ifNull** Value to return if the number is NULL.

**status** Status of conversion:
- -2 NULL
- -1 truncated or negative
- 0 ok
- 1 overflow

## toFloat/toDouble

Convert a number to a 4- or 8-byte floating point value. If no parameter is given and the number is NULL, then 0.0 is returned.

```
float toFloat();
float toFloat(float ifNull);

double toDouble();
double toDouble(double ifNull);
```

**ifNull** Value to return if the number is NULL.

## sqrt

Return the square root of a number.

```
dbNumber *sqrt();
```

# Chapter 9
# dbObject

## class dbObject

The base class for all VORTEX++ data objects. The derived clases are:

- `dbBinary`
- `dbChar`
- `dbDate`
- `dbInt`
- `dbNumber`

## Data Members

See sub-classes.

## Constructors

See sub-classes.

## Destructor.

See sub-classes.

## Member Functions

### getDatatype/getLength

Return data type and length of the object. See `dbDescriptor.hxx` for datatype values.

```
int getDatatype() const;
int getLength()   const;
```

### getAddr

Return the address of the actual (raw) data.

```
virtual void *getAddr() const;
```

### toChar

Convert a data object to a null-terminated string. Individual sub-class may have additional conversion functions.

```
virtual char *toChar(char      *bufp,
                     const int  maxBufLength,
                     const int  mustCopy,
                     char      *ifNull);
```

**bufp** Address of target character buffer. If `mustCopy` is false then this buffer is not used for character or binary objects.

**maxBufLength** Allocated size of the target buffer.

**mustCopy** Forces the use of the target buffer.

**ifNull** A null-terminated string that is returned if the data object is NULL.

## copyObject

Make a copy of a data object.

```
virtual dbObject *copyObject();
```

# Chapter 10 dbStatement

## class dbStatement

Encapsulates a SQL statement.

## Data Members

None.

## Constructors

The 3 last constructors are convenience constructors that simply add a call to `dbChannel::sql`.

```
dbStatement(dbChannel   *theChannel);
dbStatement(dbChannel   *theChannel,
            const  char *sqlStatement);
dbStatement(dbChannel   *theChannel,
            const  char *sqlStatement,
            const  int   numParameters);
dbStatement(dbChannel   *theChannel,
            const int    isProc,
            const char  *sqlStatement,
            const int    numParameters);
dbStatement(dbChannel   *theChannel,
            const char  *sqlStatement,
            const int    numDimensions,
            const int    numParameters);
```

| | |
|---|---|
| **theChannel** | A previously instantiated `dbChannel`. |
| **sqlStatement** | A null-terminated string containing the SQL statement. |
| **isProc** | Indicates that the `sqlStatement` is a procedure. |
| **numParameters** | Number of parameters in the SQL statement. If not specified then the statement is assumed to have no paramters. |
| **numDimensions** | Number of dimensions of paramters in the SQL statement. If not specified and `numParamters` is specified, the value defaults to 1. |

## Destructor

A "hard" close on the underlying database cursor is performed before freeing all working space.

```
~dbStatement();
```

# Exceptions

The following exceptions can be thrown by dbStatement functions.

dbExceptError

Invalid method for setting stored procedure parameters

Row n out of bounds (range: 0 - m)

Column n out of bounds (range: 0 - m)

Cannot mix copyParam() with prior setParam()

Cannot mix setHostVar(), copyParam(), and setParam()

Index n out of bounds (range: 0 - m)

Only valid for setting stored procedure parameters

# Member Functions

These functions are listed in `dbStatement.hxx`.

## hasBlobs

Indicates that the statement contains BLOBs. When fetching data this ensures that only one row at a time is fetched from the database.

```
void hasBlob();
```

## execute

Execute a non-SELECT statement. Convenience functions for `dbChannel::execute`.

```
void execute();
void execute(const int ingoreDbError);
```

## fetch

Execute the first fetch for a SELECT statement. Convenience function for `dbChannel::fetch`.

```
void fetch();
```

## close

Close a statement (cursor). Convenience function for `dbChannel::close`.

```
void close(const int hardClose);
```

## getBlob

Get a BLOB from a fetch statement. Convenience function for `dbChannel::getBlob`.

```
unsigned int getBlob(int           column,
```

```
                        char         *buffer,
                        unsigned int  bufferLength);
```

## putBlob

Put (send) a BLOB to a database for storage. Convenience function for `dbChannel::putBlob`.

```
void putBlob(int           isBinary,
             char         *buffer,
             unsigned int  bufferLength);
```

## nextObject

Retrieve the next column from a fetch statement. Convenience function for `dbChannel::nextObject`.

```
dbObject *nextObject();
int nextObject(dbObject *returnObject,
               int       maxLength);
```

## skipObject

Skip objects from a fetch statement. Convenience function for `dbChannel::skipObject`.

```
void skipObject(int       numColsToSkip,
                const int stopOnColZero);
```

## copyParam

Copy a parameter to the internal parameter buffer. An actual copy of the parameter is made.

```
void copyParam(int       row,
               int       column,
               int       value);
void copy Param(int      row,
               int       column,
               dbObject *value);
```

**row** The zero-based destination row index.

**column** The zero-based destination column index.

**value** The parameter.

## setParam

Make a reference to the parameter in the internal parameter buffer. The caller is responsible for freeing the parameter, if its necessary.

```
void setParam(int        row,
```

```
                int       column,
                dbObject *value);
```

| | |
|---|---|
| **row** | The zero-based destination row index. |
| **column** | The zero-based destination column index. |
| **value** | The parameter. |

## setHostVar

Make a reference to the parameter in the internal parameter buffer. The caller is responsible for freeing the parameter, if it's necessary. The first version is used for setting parameters in regular SQL statements. The second version is only used for stored procedure statements.

```
void setHostVar(int  index,
                int  datatype,
                int   length,
                void  *variable,
                short *indicator);

void setHostVar(int   index,
                int   datatype,
                int   length,
                int   items,
                int   input,
                int   output,
                char  *name,
                void  *variable,
                short *indicator);
```

| | |
|---|---|
| **index** | The zero-based parameter index. |
| **datatype** | The variable's data type. |
| **length** | The variable's length. |
| **items** | 1 = scalar<br>> 1 = array |
| **input** | Is an input variable. |
| **output** | Is an output variable. |
| **name** | A null-terminated string containing the variable's name. |
| **variable** | Address of the variable. |
| **indicator** | Address of the indicator variable. You can specify a NULL. |

## numOutputCols

Return the number of output columns in the SELECT statement.

```
int numOutputCols();
```

## getColDesc

Return a description of an output column. It is the caller's responsibility to free the `dbDescription`.

```
dbDescriptor *getColDesc(int column);
```

**column** The zero-based column index.

## endOfScan

Return true of an "end of scan" has been reached on a SELECT statement.

```
int endOfScan() const;
```

## getChannel

Return the `dbChannel` to which the statement belongs.

```
dbChannel *getChannel() const;
```

## T



## U



## V



## W



## X